



Pp. 283–298 of the *Proceedings*

The Elements of Cache Programming Style

Chris B. Sears

Google Inc.

Those who cannot remember the past are condemned to repeat it - George Santayana

Introduction

Cache memories work on the carrot and stick principle. The carrot is the Principle of Locality and the stick is Amdahl's Law. The Principle of Locality says that programs tend to cluster their memory references. A memory location referenced once is likely to be referenced again: temporal locality. A memory location nearby a referenced location is likely to be referenced soon: spatial locality. And Amdahl's Law says that the performance improvement to be gained from using some faster component is limited by the fraction of time the faster component is used. In this case, CPU and cache are fast components and memory is slow.

If your program adheres to the Principle of Locality, it benefits from fast cache memory and runs at processor speed. If it doesn't, it is held accountable to Amdahl's Law and runs at memory speed. Hit rates have to be very high, say 98%, before incremental increases in processor speed are even noticeable.

Amdahl's Law has a special circumstances penalty for multiprocessors [Schimmel94]. Thrashing on a multiprocessor can slow down all of the processors. They each wait for each other waiting for memory and the leverage a multiprocessor offers works in reverse. Adherence to the Principle of Locality for multiprocessors, but not to the point of False Sharing, isn't just a nicety, it is a necessity.

The object of cache programming style is to increase this locality. It is important to understand the structure and behavior of caches, but it is more important to know the basic properties to take advantage of and the worst cases to avoid. This article goes into details and the summary provides guidelines.

This page
is legacy
content.



Check out the current
u s e n i x
Web site.

An Example

As a running example, I am going to look at Linux [Maxwell99] and at the scheduler in particular. The idea is to modify data structures and code just slightly, trying to use the cache more effectively. Hopefully I will achieve two goals: a practical tutorial on caches and some performance improvements for Linux.

Instead of talking about cache memory systems in general, I will mostly use my circa 1998 350 MHz Deschutes Pentium II system as a specific example. It has these characteristics:

Storage	Size	Latency	Notes
register	32 bytes	3 ns	register renaming file
L1	32 K	6 ns	on-chip, half Pentium-II clock rate
L2	256 K	57 ns	off-chip, on-package [Intel99a]
memory	64 MB	162 ns	100 MHz SDRAM, single bank
disk	10 GB	9 ms	DMA IDE
network	whatever	whenever	56K PPP

Figure 1. Storage hierarchy sizes and latencies

These numbers are subject to change. CPU performance improves at about 55%/year and memory improves at about 7%/year. Memory is big, cheap and slow while cache is small, fast and expensive. Double Data Rate SDRAM and Rambus, when available, will improve memory bandwidth but not latency. These improvements will help more predictable applications like multimedia but not less predictable programs such as Linux.

Cache Basics

First, a few words about caches in general. Cache memory fits into the storage hierarchy in terms of both size and speed. Cache line misses, page faults and HTTP requests are the same thing at different levels of this hierarchy. When a Squid proxy doesn't have an object in its cache, it forwards the HTTP request to the origin server. When a CPU requests an address which isn't in memory, a page fault occurs and the page is read from disk. When a CPU requests an address which isn't in cache, the containing cache line is read from memory. LRU, working set, associative, coherency, hashing, prefetching are all techniques and terminology which are used in each level of the storage hierarchy.

In each case, one smaller faster level in the hierarchy is backed by another bigger slower level. If performance is limited by excessive use of the slower level, then according to Amdahl's Law, little improvement can be made by just making the faster level faster.

With respect to cache memory [Handy98], the most important thing to understand is the cache line. Typically a cache line is 32 bytes long and it is aligned to a 32 byte offset. First a block of memory, a memory line, is loaded into a cache line. This cost is a cache miss, the latency of memory. Then,

after loading, bytes within a cache line can be referenced without penalty as long as it remains in the cache. If the cache line isn't used it will be dropped eventually when another memory line needs to be loaded. If the cache line is modified, it will need to be written before it is dropped.

This is the simplest and most important view of a cache memory. Its lesson is two-fold: pack as much into a cache line as possible and use as few cache lines as possible. Future memory bandwidth increases (DDR and Rambus) will reward this practice. The more complex characteristics of cache, the structure and behavior, are important for understanding and avoiding worst case cache behavior: thrashing.

Competing for and sharing of cache lines is a good thing, up to a point, when it becomes a bad thing. Ideally a fast cache will have a high cache hit rate and the performance will not be bound to the speed of the memory. But a really bad thing, thrashing, happens when there is too much competition for too few cache lines. This happens in worst case scenarios for data structures. Unfortunately the current profiling tools look at the instructions rather than data. This means that a programmer must be aware of worst case scenarios for data structures and avoid them. A useful tool for finding a hot spot is cacheprof [Seward].

The Pentium II L1 and L2 Caches

The Pentium II [Shanley97] 32K L1 cache consists of 1024 32 byte cache lines partitioned into instruction and data banks of 512 lines each. It uses the color bits 5-11 to index into an array of sets of cache lines. In parallel, it compares the tag bits 12-31 (12-35 with Pentium III Physical Address Extension) for each of the cache lines in the indexed set. L1 uses a 4-way set associative mapping which divides the 512 lines into 128 sets of 4 cache lines.

Each of these sets is really a least recently used (LRU) list. If there is a match, the matching cache line is used and it is moved to the front of the list. If there isn't a match, the data is fetched from L2, the cache line at the end of the list is replaced and the new entry is put at the front of the list.

Two memory lines of the same color compete for the same set of 4 L1 cache lines. They are off the same color if their color bits (5-11) are the same. Alternatively they are of the same color if their addresses differ by a multiple of 4096: $2^{(7 \text{ color bits} + 5 \text{ offset bits})}$. For example, address 64 and 12352 differ by 12288 which is 3×4096 . So, 64 and 12352 compete for a total of 4 L1 cache lines. But 64 and 12384 differ by 12320, not a multiple of 4096, so they don't compete for the same L1 cache lines.

Instructions are also cached. The Pentium II L1 cache is a Harvard, or split instruction/data cache. This means that instructions and data never compete for the same L1 cache lines. L2 is a unified cache. Unified means that there is a single cache bank and that instructions and data compete for cache lines.

L2 is similar to L1 except larger and much slower. The on-package 256K L2 cache on my Pentium II has 8192 cache lines. It is also 4-way set associative but is unified. There are Pentium II's with 512K of L2 which increase the set size to 8. Also, there are PIII's with up to 2 MB of L2. If there is a cache line miss for L2, the cache line is fetched from memory. Two memory lines compete for the same L2 cache lines if they differ by a multiple of 64K: $2^{(11 \text{ cache color bits} + 5 \text{ offset bits})}$.

The important things to remember about my Pentium II are:

-
- cache lines are 32 bytes in size and are aligned to 32 byte offsets
- memory locations which are offset by multiples of 4K bytes compete for 4 L1 cache lines

- memory locations which are offset by multiples of 64K bytes compete for 4 L2 cache lines.
- L1 has separate cache lines for instructions and data - Harvard
- L2 shares cache lines between instructions and data - unified

Variable Alignment

We will start with the simple stuff. It is better to align just about everything to a long word boundary. Linux is written in the gcc programming language and a careful study of the gcc standards document, "Using and Porting GNU CC" [Stallman00], is therefore necessary: no one embraces and extends quite like Richard Stallman. gcc is particularly helpful with structure field alignment which are intelligently and automatically aligned. ANSI C Standard allows for packing or padding according to the implementation.

```
struct dirent {
    long      d_ino;
    __kernel_off_t d_off;
    unsigned short d_reclen;
    char       d_name[256];
};
```

Figure 2. <linux/dirent.h>

gcc automatically aligns d_reclen to a long boundary. This works well for unsigned short, but for short on the x86 the compiler must insert sign extension instructions. If you are using a short to save space, consider using an unsigned short. For example, in <linux/mm.h> changing the field vm_avl_height into an unsigned short saves 32 bytes of instructions for a typical build. It could just as well be an int.

```
struct vm_area_struct {
    ...
    short      vm_avl_height;      // unsigned short or int
    ...
};
```

Figure 3. struct vm_area_struct

Strings should be aligned as well. For example, strncmp() can compare two long words at a time, cheap SIMD, if both source and destination are long word aligned. The x86 code generator for egcs 2.95.2 has a nice little bug that doesn't align short strings at all and aligns long strings to the cache line:

```
char* short_string = "a_short_string";
char* long_string = "a_long_long_long_long_long_long_string";

.LC0:
    .string      "a_short_string"      // an unaligned string
```

```

...
.align 32
.LC1:                                     // aligned to cache line
.string    "a_long_long_long_long_long_long_string"

```

Figure 4. GCC x86 string disassembly

What is necessary here is to align both strings to long words with `.align 4`. This uses less space and has better alignment. On a typical Linux build, this saves about 8K.

Cache Alignment of Structures

Arrays and lists of structures offer an opportunity to cache align large amounts of data. If the frequently accessed fields are collected into a single cache line, they can be loaded with a single memory access. This can reduce latency and cache footprint. However, it can also increase cache footprint if large amounts of data are being accessed. In this case, packing efficiency and also cache pollution are more important.

So for arrays, the base of an array should be cache aligned. The size of a structure must be either an integer multiple or an integer divisor of the cache line size. If these conditions hold, then by induction, each element of the array the cache line will be aligned or packed. Linked structures are analogous for alignment but don't have the size constraint.

An array of structures of type `mem_map_t` is used by the page allocator as a software page table:

```

/*
 * Try to keep the most commonly accessed fields in single cache lines
 * here (16 bytes or greater). This ordering should be particularly
 * beneficial on 32-bit processors. ...
 */
typedef struct page {
    struct list_head    list;           // from linux-2.4.0-test2
    struct address_space* mapping;      // 2,4
    unsigned long       index;          // 1,2
    struct page*        next_hash;      // 1,2
    atomic_t            count;          // 1,1+1
    unsigned long       flags;          // 1,2
    struct list_head    lru;            // 2,4
    wait_queue_head_t   wait;           // 5,10
    struct page**       pprev_hash;     // 1,2
    struct buffer_head* buffers;        // 1,2
    unsigned long       virtual;        // 1,2
    struct zone_struct* zone;           // 1,2
} mem_map_t;
// 18 * 4 == 72 x86
// 36 * 4 == 144 Alpha

```

Figure 5. mem_map_t from <linux/mm.h>

On a 32-bit Pentium, the size of mem_map_t is 72 bytes. It was 40 bytes in 2.2.16. Since the array allocation code uses sizeof(mem_map_t) to align the array, the base is aligned incorrectly as well. In any case MAP_ALIGN() can be replaced with L1_CACHE_ALIGN() which uses simpler code:

```
#define MAP_ALIGN(x) (((x) % sizeof(mem_map_t)) == 0)          \
    ? (x) : ((x) + sizeof(mem_map_t) - ((x) % sizeof(mem_map_t)))

lmem_map = (struct page *) (PAGE_OFFSET +
    MAP_ALIGN((unsigned long) lmem_map - PAGE_OFFSET));

#define L1_CACHE_ALIGN(x) (((x) + (L1_CACHE_BYTES - 1))      \
    & ~(L1_CACHE_BYTES - 1))

lmem_map = (struct page *) L1_CACHE_ALIGN((unsigned long) lmem_map);
```

Figure 6. lmem_map alignment

On a 64-bit Alpha, a long is 8 bytes with an 8 byte alignment and sizeof(mem_map_t) is 144 bytes. The flags field doesn't need to be a long, it should be an int. Since atomic_t is also an int and the two fields are adjacent, they would pack into a single long word. The page wait queue head used to be a pointer. Changing it back would save enough to allow cache aligning both 32-bit and 64-bit versions.

Cache Line Alignment for Different Architectures

It is possible to target and conditionally compile for a particular processor. Linux has an include file, <asm-i386/cache.h>, defining the L1 cache line size, L1_CACHE_BYTES, for the x86 architecture family. The slab allocator [Bonwick94], which allocates small objects from memory pages, uses L1_CACHE_BYTES when a client requests a cache aligned object with the SLAB_HWCACHE_ALIGN flag.

```
/*
 * include/asm-i386/cache.h
 */
#ifndef __ARCH_I386_CACHE_H
#define __ARCH_I386_CACHE_H
/* bytes per L1 cache line */
#if CPU==586 || CPU==686
#define L1_CACHE_BYTES 32
#else
#define L1_CACHE_BYTES 16
#endif
#endif
```

Figure 7. <asm-i386/cache.h>

If someone got a Red Hat kernel conservatively compiled targeting the 486, then it assumed 16 byte cache lines. It was also wrong for the Athlon. This has been fixed in 2.4 by defining and using the kernel configuration macro `CONFIG_X86_L1_CACHE_BYTES` in `<linux/autoconf.h>`.

If you must assume one cache line size when laying out the fields inside of structs intended for portable software, use 32 byte cache lines. For example, `mem_map_t` could use this. Notice that 32 byte aligned cache lines are also 16 byte aligned. The PowerPC 601 nominally has a 64 byte cache line but it really has two connected 32 byte cache lines. The Sparc64 has a 32 byte L1 and a 64 byte L2 cache line. It is much easier to think of all systems as having 32 byte cache lines and enumerate the exceptions, if any. Alpha and Sparc64 have 32 byte cache lines but the Athlon and Itanium, the exceptions that proves the rule, have 64 byte cache lines. And the IBM S/390 G6 has a 256K L1 cache with 128 byte cache lines.

On the vast majority of processors, 32 byte cache lines is the right thing to do. And most importantly, if you have addressed and avoided the footprint and worst case thrashing scenarios in the 32 byte case, you will have avoided them for the other cases.

Caching and the Linux Scheduler

Linux represents each process with a `task_struct` which is allocated two 4K pages. The task list is a list of the `task_struct`'s of all existing processes. The runqueue is a list of the `task_struct`'s of all runnable processes. Each time the scheduler needs to find another process to run, it searches the entire runqueue for the most deserving process.

Some folks at IBM [Bryant00] noticed that if there were a couple of thousand threads that scheduling took a significant percentage of the available CPU time. On a uniprocessor machine with a couple of thousand native Java threads, just the scheduler alone was taking up more than 25% of the available CPU. This gets worse on a shared memory SMP machine because memory bus contention goes up. This doesn't scale.

It turned out that the `goodness()` routine in the scheduler referenced several different cache lines in the `task_struct`. After reorganizing `task_struct`, `goodness()` now references only a single cache line and the CPU cycle count was reduced from 179 cycles to 115 cycles. This is still a lot.

Here is the important cache line, the Linux scheduling loop and the `goodness()` routine. The scheduler loop iterates through the entire runqueue, evaluates each process with `goodness()` and finds the best process to run next.

```
struct task_struct {
    ...
    long          counter;           // critical 2cd cache line
    long          priority;
    unsigned long policy;
    struct mm_struct *mm, *active_mm;
    int           has_cpu;
    int           processor;
    struct list_head run_list;       // only first long word
```

```

    ...
};

tmp = runqueue_head.next;
while (tmp != &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p)) {           // running on another CPU
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
    tmp = tmp->next;
}

#define PROC_CHANGE_PENALTY 15        // processor affinity

static inline int goodness(struct task_struct *p,
    int this_cpu, struct mm_struct *this_mm)
{
    int weight;
    if (p->policy != SCHED_OTHER) {
        weight = 1000 + p->rt_priority; // realtime processes
        goto out;
    }
    weight = p->counter;
    if (!weight)
        goto out;                    // no quanta left
#ifdef __SMP__
    if (p->processor == this_cpu)
        weight += PROC_CHANGE_PENALTY; // processor affinity
#endif
    if (p->mm == this_mm)             // same thread class
        weight += 1;                  // as current?
    weight += p->priority;
out:
    return weight;
}

```

Figure 8. task_struct and scheduler loop

A long runqueue is certainly not the common case even for heavily loaded servers. This is because event driven programs essentially self schedule with poll(). The contrasting style, threading, is preferred by Java, Apache and TUX. It is ironic that poll() also had scalability problems, and on other Unix systems as well [Honeyman99]. Also, the Linux 2.4 x86 kernels increase the maximum number of threads past 4000.

On SMP machines, processes have a scheduling affinity with the last CPU they ran on. The idea is that some of the working set is still in the local cache. But the scheduler has a subtle SMP bug. When a CPU has no processes on the runqueue, the scheduler will assign it a runnable process with an affinity

to another CPU. It would be wiser to first dole out more quanta to processes on the runqueue, perhaps only those with an affinity to that CPU. Even then it may be better to idle, particularly with a short runqueue.

Cache Line Prefetching

Modern CPUs aggressively prefetch instructions but what about data? CPUs don't prefetch data cache lines, but vectorizing compilers do and programs can. Depending on the amount of CPU processing per cache line, you may need to prefetch more than one cache line ahead. If the prefetch is scheduled sufficiently far in advance, it won't matter if the cache line is in memory rather than L2 [Intel99a].

Typically prefetching is used in multimedia kernels and matrix operations where the prefetched address can be easily calculated. Algorithms operating on data structures can use prefetch as well. The same methods apply except that the prefetched address will follow a link rather than an address calculation. Prefetching for data structures is important since memory bandwidth is increasing faster than latency is decreasing. Traversing a data structure is more likely to suffer from a latency problem. Often only a few fields in a structure are used whereas with multimedia usually every bit is examined.

Prefetching From Memory

If a prefetch instruction can be scheduled 20-25 or so instructions before the cache line will be used, the fetch can completely overlap instruction execution. The exact prefetch scheduling distance is a characteristic of the processor and memory. Superscalar processors execute more than one instruction at a time.

Prefetching From L2

If an algorithm is traversing a data structure likely to be in L2, and it can schedule a prefetch 6-10 instructions before the cache line will be used, the fetch can completely overlap instruction execution.

The Linux scheduler loop is a good candidate for cache line prefetching from L2 because goodness() is short and after the IBM patch, it only touches a single cache line.

Here is a prefetching version of the scheduler. It overlaps the prefetch of the next cache line from L2 during the execution of goodness().

```
tmp = runqueue_head.next;
while (tmp != &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    tmp = tmp->next;
    CacheLine_Prefetch(tmp->next);          // movl xx(%ebx),%eax
    if (can_schedule(p)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
```

```

        c = weight, next = p;
    }
}

```

Figure 9. Prefetching scheduler loop

By the way, moving the tmp pointer chase before the goodness() call ends up using fewer instructions than the original. And with a little effort, the loop could omit tmp as well.

```

inline void CacheLine_Prefetch(unsigned long addr)
{
    asm volatile("" : : "r" (addr));
}

```

Figure 10. CacheLine_Prefetch()

This little bit of gcc magic is actually architecture-independent assembly code. It basically means load from addr into some temporary register of the compiler's choosing. So technically, I lied. It isn't a prefetch, it's really a preload. A prefetch offers more cache control but has some restrictions.

CacheLine_Prefetch() should be specialized on different architectures to take advantage of the various prefetch instructions. In fact, CacheLine_Prefetch() should be wrapped in conditional compilation logic because it may be inappropriate on certain Early Bronze Age machines. Also, AMD uses a slightly different set of prefetch instructions not strictly based on the MMX. On the Pentium II, this could be:

```

inline void CacheLine_Prefetch(unsigned long addr)
{
    asm volatile("prefetcht0 (%0)" :: "r" (addr));
}

```

Figure 11. Pentium II CacheLine_Prefetch()

Caches and Iterating Through Lists

When repeatedly iterating through an array or a list of structures, be careful of cache considerations. As the number of elements increases and approaches the number of cache lines available, thrashing will gradually increase. The gradually increasing thrashing is why this performance problem is hard to find.

The Linux scheduler iterates through the runqueue to find the next process to run. Linux also uses for_each_task() to iterate through each task_struct on the task list and perform some work. Iterating through lists represents potentially large amounts of memory traffic and cache footprint. Here is the for_each_task() iterator macro:

```

#define for_each_task(p) \
    for (p = &init_task ; (p = p->next_task) != &init_task ; )

```

Figure 12. for_each_task() iterator

for_each_task() can be combined with CacheLine_Prefetch(). Notice that for_each_task() uses next_task which isn't in the preferred cache line. This doubles the memory traffic and cache footprint.

```

#define for_each_task(p) \
    for (p = &init_task ; p = p->next_task, \
         CacheLine_Prefetch(p->next_task), \
         p != &init_task ; )

```

Figure 13. prefetching for_each_task() iterator

As an example, when all of the processes on the runqueue have used up their scheduling quanta, Linux uses for_each_task() to dole out more:

```

recalculate:
{
    struct task_struct *p;
    spin_unlock_irq(&runqueue_lock);
    read_lock(&tasklist_lock);
    for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
    read_unlock(&tasklist_lock);
    spin_lock_irq(&runqueue_lock);
}
goto repeat_schedule;

```

Figure 14. Scheduler recalculate loop

As an observation, recalculate should iterate through the runqueue instead of the task queue. It is shorter and there is no reason to dole out more quanta to sleeping processes. counter for sleeping processes will grow without bound. When the sleeping process wakes up, it may have a large amount of quanta stored up, disturbing the responsiveness of the other processes. FreeBSD [McKusick96] recomputes the scheduling priority if an awoken process was sleeping for more than a second.

Linux also uses for_each_task() occasionally when allocating a process id in get_pid():

```

for_each_task(p) {
    if(p->pid == last_pid ||
        p->pgrp == last_pid ||

```

```

        p->session == last_pid) {
            ...
        }
    }
}

```

Figure 15. pid allocation loop

Examining the uses of `for_each_task()` it is possible to change the critical `task_struct` cache line. Some fields are moved out, some need to be added and some need to be made smaller. `active_mm` is an unnecessary field for purposes of the scheduler loop. The kernel type `pid_t` is currently a 4 byte int. However, the maximum value for a process id is `PID_MAX` which is `0x8000`. So `pid_t` can be changed to an unsigned short. (NB, `PID_MAX` will probably increase in 2.5). `priority` is limited to the range `0..40` and `counter` is derived from it. `policy` is restricted to six values. `processor` is currently an int and `NR_CPUS` is 32 so changing it to an unsigned short is reasonable.

After these changes, several of the uses of `for_each_task()` as well as the process id hash, `find_task_by_pid`, restrict their references to a single cache line.

```

struct task_struct {
    ...
    unsigned char counter;           // beginning
    unsigned char priority;
    unsigned char policy_has_cpu;
    /* one char to spare */         // one
    unsigned short processor;
    unsigned short pid;             // two
    unsigned short pgrp;
    unsigned short session;         // three
    struct mm_struct *mm;           // four
    struct task_struct *next_task;  // five
    struct task_struct *pidhash_next; // six
    struct task_struct *run_next;   // seven
    /* one long word to spare */    // eight
    ...
};

```

Figure 16. Packed task_struct for scheduler

If possible, squeeze everything the processing loop uses into as few cache lines as possible, preferably just one. In your data structures, if you have to use short instead of int, use short. If it will make it fit, use nibbles and bits. If you can't say it in 32 bytes, perhaps you need to rephrase yourself.

Thrashing

OK, you might want to close your eyes about now because it's going to start getting really gory. Now we look at the worst case scenarios of thrashing. It also gets more complex.

The task_struct is allocated as two pages which are 4K aligned. The L1 cache divides the 32-bit address space among the 128 groups each with a set of 4 cache lines. If two addresses are separated by a multiple of 4K bytes, then they map to the same cache line set. So for the 4-way set associative L1 cache on a Pentium II there are 4 cache lines available for the scheduling related cache line for all of the task_structs.

Really that isn't so bad. Asking a task_struct cache line to remain in L1 for a schedule quantum is, I admit, a bit much. But the situation doesn't really improve for L2. A 256K L2 cache will provide only 64 suitable cache lines. The way this works is that a task_struct is 4K page aligned. So in the L2 cache set index, bits 5-11 will be fixed. So there are 4 bits of possibly unique cache set index, or 16 sets of 4 cache lines or 64 available cache lines.

Furthermore, L2 is managed LRU. If you are iterating through a runqueue longer than the effective L2 set depth of 64, when you exceed the set size of L2, the cache thrashes and you may as well have no cache. Then, every reschedule the scheduler is scanning the entire runqueue from memory. Prefetching is rendered useless. Also, the set is not a list of 64 cache lines but really 16 associative subsets of 4 cache lines. Thrashing begins gradually before reaching the set size of 64.

But it gets worse. I forgot to tell you about the TLB. Logical addresses are mapped to physical addresses via a two level page table in memory and an on-chip memory management unit.

```
virtual address
  page group - bits 22-31
  page address - bits 12-21
  page offset - bits 0-11

physical page lookup
  table_level_2 = table_level_1[(vaddr & mask1) >> shift1];
  page         = table_level_2[(vaddr & mask2) >> shift2];
```

Figure 17. Pentium II VM address structure and translation

These mappings are expensive to compute. The masks and shifts are free but the loads and adds are serial. The Alpha and Itanium use a 3 level page table structure. Some page tables can be so large they must be demand paged. But since the mappings rarely change, they are computed on demand and the results are cached in the TLB. The Pentium II Translation Lookaside Buffer (TLB) uses a Harvard 4-way set associative cache with 64 data entries and 32 instruction entries. The TLB replacement policy is LRU within the set. If the TLB doesn't find a match, a TLB miss takes place. The cost of a TLB miss ranges from a reported 5 cycles on a Xeon, to 60 or more cycles. On the PowerPC it is handled in software. The PowerPC TLB miss handler is 29 instructions.

Virtual memory pages are 4K. A given task_struct page will compete for 4 TLB slots. Iterating through a long linked list (64) of 4K aligned pages is really the worst case scenario for the TLB. The iteration will flush the TLB, slowly, suffering TLB misses along the way. But for the purposes of the runqueue, the TLB could just as well have no entries at all. Linux invalidates the TLB at each process context switch. Then, every entry on the runqueue will be a TLB miss regardless of the size of the runqueue. The only case where this doesn't happen is if a process context switches to itself. Even in this case, a long runqueue can thrash and flush the TLB cache. Also, the TLB miss is synchronous with the cache line miss and a TLB miss will cause the prefetch to be ignored.

Avoid iterating through collections of 4K pages. It is a worst case cache and TLB scenario. Actually it is a member of a dysfunctional family of worst case scenarios. If addresses differ by a power of the cache line size, 64, 128, 256, ..., they compete for cache lines. Each increase in the exponent halves the number of cache lines available, down to the minimum of the cache set size. If addresses differ by a power of the page size, 8192, 16384, ..., they compete for diminishing TLB slots. And while it is possible to add more L2 on some systems, this isn't possible for TLB slots.

Pollution

An application can use prefetching to overlap execution with memory fetch. But once referenced, some memory is used only once and shouldn't evict more important cache lines and shouldn't take up space in the cache. If prefetch determines what cache lines will be needed, the cache control instructions determine what cache lines will be retained. With respect to performance, avoiding cache pollution is as important as prefetch.

As an example, there is a problem with the cache line oriented task_struct. The for_each_task() macro iterates through every task, polluting the L2 cache, flushing good data and loading junk. The junk data will mean further cache misses later.

On the Pentium II there is the non-temporal prefetchnta instruction. This loads a cache line into L1. If it is already in L2, it is loaded from L2. But if it isn't in L2, it isn't loaded into L2. The PowerPC doesn't have this sort of instruction. On that processor, prefetch first and then cache flush a junk cache line after using it. A junk cache line in this case would be a process not on the runqueue. This reduces L2 cache pollution but doesn't avoid it altogether as in the Pentium II prefetchnta example.

```
inline void CacheLine_Flush(unsigned long addr)
{ /* no cache line flush on the x86 */ }

inline void CacheLine_NT_Prefetch(unsigned long addr)
{
    asm volatile("prefetchnta (%0)" :: "r" (addr));
}

#define for_each_task(p) \
    for (p = &init_task; p->next ? 1 : CacheLine_Flush(p), \
         p = p->next_task, CacheLine_NT_Prefetch(p->next_task), \
         p != &init_task;)
```

Figure 18. Pollution avoiding prefetching

Interrupt handlers and basic functions such as copying i/o buffer cache pages to and from user space with memcpy() would benefit from non-temporal prefetch, cache flush and streaming store instructions to avoid polluting L2. The Linux 2.4 x86 _mmx_memcpy() prefetches source cache lines. For a Pentium II, it should use prefetchnta for both the source and destination in order to avoid flushing and polluting L2.

On the Pentium II, an initial preload is necessary to prime the TLB for any following prefetch instructions. Otherwise the prefetch will be ignored. This

is another argument against iterating through lists or arrays of 4K structures: a preload is necessary to prime the TLB but the preload will pollute the cache and there is no cache line flush instruction on the Pentium II.

As an example, this is a Pentium II user mode cache line block copy for up to a 4K page. This function assumes that the source and destination will not be immediately reused. The `prefetchnta` instructions marks the source and destination cache lines as non temporal. A Pentium III version [Intel99b] can use `movntq` and the streaming instructions. However, the streaming instructions require additional OS support for saving the 8 128-bit Katmai registers at context switch. Patches are available for 2.2.14+ [Ingo00].

```
void PII_BlockCopy(char* src, char* dst, int count)
{
    char    *limit;

    asm volatile("" :: "r" (*src)); // prime the TLB for prefetch
    asm volatile("" :: "r" (*dst));
    asm volatile("" :: "r" (*(src + 4095))); // src may span page
    asm volatile("" :: "r" (*(dst + 4095)));

    for (limit = src + count; src < limit; src += 32, dst += 32) {
        asm volatile("prefetchnta (%0)" :: "r" (src));
        asm volatile("prefetchnta (%0)" :: "r" (dst));
        asm volatile("movq 00(%0),%%mm0" :: "r" (src));
        asm volatile("movq 08(%0),%%mm1" :: "r" (src));
        asm volatile("movq 16(%0),%%mm2" :: "r" (src));
        asm volatile("movq 24(%0),%%mm3" :: "r" (src));
        asm volatile("movq %%mm0,00(%0)" :: "r" (dst));
        asm volatile("movq %%mm1,08(%0)" :: "r" (dst));
        asm volatile("movq %%mm2,16(%0)" :: "r" (dst));
        asm volatile("movq %%mm3,24(%0)" :: "r" (dst));
    }
    asm volatile("emms"); // empty the MMX state
}
```

Figure 19. PII_BlockCopy()

Another candidate is `memset()`. Idle task page clearing with `memset()` has been tried but it isn't done because of cache pollution. Memory stores fill up cache lines just as memory loads do. But cache pollution can be avoided by `prefetchnta` of the destination cache line followed by the store. `prefetchnta` a priori tags the destination cache line as non cacheable. Another alternative on the PowerPC is marking the page to be cleared as non-cacheable but this is privileged [Dougan99].

False Sharing

A variation on the theme of thrashing is False Sharing [HennPatt96]. Two variables contained in a single cache line are updated by two different CPUs on a multiprocessor. When the first CPU stores into its variable in its cache line copy, it invalidates the cache line copy in the second CPU. When the

second CPU stores into its variable, reloads the cache line, stores into it and invalidates the cache line copy in the first CPU. This is thrashing. Allocating each variable its own cache line solves this problem.

An example from the scheduler, showing the problem and solution, is the current task array variable:

```
struct task_struct *current_set[NR_CPUS];    // Linux 2.0.35

static union {                               // Linux 2.2.14
    struct schedule_data {
        struct task_struct * curr;
        cycles_t last_schedule;
    } schedule_data;
    char __pad [SMP_CACHE_BYTES];
} aligned_data [NR_CPUS] __cacheline_aligned = { {{&init_task,0}}};
```

Figure 20. SMP scheduler global data array

Page Coloring

Two memory lines separated by modulo 64K compete for just 4 L2 cache lines. Within this 64K span, memory lines do not compete. Breaking this 64K up into 16 4K memory pages, each has a unique page color. Physical memory pages of different color don't compete for cache.

Ideally if two virtual memory pages will be used at the same time, they should be mapped to physical pages of different color [Lynch93]. In particular, simple direct mapped caches only have a single suitable cache line. Pages 0,1,2,3 are in contiguous order and of different page color. Pages 16,1,18,35 also each have different color. Page coloring also improves performance repeatability.

Higher levels of L2 associativity argue against the expense of supporting page coloring in the OS. Linux does not currently support page colored allocation because it would be too expensive. FreeBSD attempts to allocate pages in color order and there has been discussion of this for Linux if an inexpensive page allocation approach can be found.

However, page coloring is supported in one special case: `__get_dma_pages()`. This kernel function allocates up to 32 physically contiguous pages. Therefore pages in color order. A hint flag for the `mmap()` system call could request this.

Constraints

Memory lines within aligned structures, perhaps aligned within pages, are constrained. The greater the alignment constraint, the fewer eligible cache lines. Constraints cause conflict cache misses.

For example, in the `task_struct` case the important cache line is competing for 64 entries. This is known as a hot cache line: it has a constraint problem. It

would be better for scheduling purposes to prune off the scheduling information and set up a slab allocated cache. The task_struct already has several data structures hanging off of it to support sharing data structures among related threads. This would be one more.

A slab is a page from which objects are allocated and freed. If a slab is full, another is constructed and the object is allocated from it. Iterating through dynamic structures allocated from slabs suffers fewer TLB misses because the structures are packed into pages.

For larger structures, frequently accessed fields are often clumped together, usually at the beginning of the structure, causing a constraint problem. Since slabs are page aligned, the slab allocator balances the cache load transparent to its clients. An offset which is a multiple of the cache line size is added as a bias.

Back To The Future

One solution for the scheduler is fairly simple: for an aligned array of structures, if the size of the structure is an odd multiple of the cache line size, it won't have a constraint problem.

Here is a single cache line version (one is an odd number) of the critical scheduling fields:

```
struct proc {
    unsigned char    priority;
    unsigned char    policy_has_cpu;
    unsigned short   processor;    // one
    unsigned short   pid;
    unsigned short   pgrp;         // two
    unsigned short   session;      // three - spare short
    struct mm_struct* mm;          // four
    struct proc*     next_task;    // five
    struct proc*     pidhash_next; // six
    struct proc*     run_next;    // seven
    struct task_struct* task_struct; // eight
};
```

Figure 21. Single cache line task_struct

Or it can be made into a two cache line structure and a few other task_struct fields can be added. If you are familiar with old Unix implementations, the cache line oriented task_struct is the reinvention of the proc structure. Quoting the 1977 Unix Version 6 <unix/proc.h> header file:

```
/*
 * One structure allocated per active
 * process.  It contains all data needed
 * about the process while the
 * process may be swapped out.
```

```

* Other per process data (user.h)
* is swapped with the process.
*/

```

Figure 22. <unix/proc.h> from 1977 Unix Version 6

The Linux scheduler searches the entire runqueue each reschedule. The FreeBSD runqueue is simpler [McKusick96]. It was derived from the 1978 VAX/VMS rescheduling interrupt handler which was all of 28 instructions. From the FreeBSD <kern/kern_switch.h> source file:

```

/*
 * We have NQS (32) run queues per scheduling class. For the
 * normal class, there are 128 priorities scaled onto these
 * 32 queues. New processes are added to the last entry in each
 * queue, and processes are selected for running by taking them
 * from the head and maintaining a simple FIFO arrangement.
 * Realtime and Idle priority processes have an explicit 0-31
 * priority which maps directly onto their class queue index.
 * When a queue has something in it, the corresponding bit is
 * set in the queuebits variable, allowing a single read to
 * determine the state of all 32 queues and then a ffs() to find
 * the first busy queue.
 */

```

Figure 23. From FreeBSD <kern/kern_switch.h>

A uniprocessor reschedule is simplicity:

```

static struct rq    queues[32];
static u_int32_t    queuebits;
int                 qi;

qi = (current->priority >> 2);
SetBit(&queuebits, qi);
InsertQueue(&queues[current->priority], current);

qi = FindFirstSet(queuebits);
if (RemoveQueue(&queues[qi], &current) == Q_EMPTY)
    ClearBit(&queuebits, qi);

```

Figure 24. Queueing scheduler context switch

A best of breed Linux scheduler would use the FreeBSD runqueue and support the Linux scheduler policies of the goodness() function: soft realtime scheduling, SMP CPU affinity and thread batching.

In Summary, You Are Doomed

Cache programming is a collection of arcana and techniques. What follows is an attempt to organize them and the final chart is an attempt to distill them.

Alignment and Packing

When laying out structures for portable software, assume a 32 byte cache line.

For arrays, the base of an array should be cache aligned. The size of a structure must be either an integer multiple or an integer divisor of the cache line size. If these conditions hold, then by induction each element of the array the cache line will be aligned or packed. Linked structures are analogous for alignment but don't have the size constraint.

To specify a cache line alignment for types or variables with gcc, use the aligned attribute:

```
struct Box {  
    int    element;  
} __attribute__((aligned(SMP_CACHE_BYTES)));
```

Figure 25. Cache alignment of structure types

In an application, to allocate cache aligned memory, use `memalign(32, size)` instead of `malloc(size)`.

Data structures change. Write a program which validates packing, alignments and worst case scenarios.

Cache Footprint

Reduce your cache footprint. Paraphrasing the Elements of Style [Strunk99]:

-
- Omit needless words.
- Keep related words together.
- Do not break cache lines in two.
- Avoid a succession of loose cache lines.
- Make the cache line the unit of composition.

When repeatedly iterating through a large array or list of structures, be careful of cache considerations. As the number of elements increases and approaches the number of cache lines available, cache misses will gradually increase to the point of thrashing.

Ignoring page color issues for now, large memory scans approaching the size of the cache, flush the cache. Large memory copies approaching half of the cache, flush the cache. Both pollute the cache with junk.

Avoid polluting the cache with junk data. If a cache line won't be reused for some time, flush it. If streaming through a lot of data, non-temporally prefetch source and destination to avoid polluting the L2 cache. Otherwise you may pollute the cache with junk data entailing further cache misses.

Interrupt handlers should avoid cache pollution by using non-temporal prefetch, cache flush and streaming store instructions.

Shared libraries allow many processes to share the same instruction memory. This creates more opportunities for cache sharing and less cache competition.

Prefetch

If you are working on one cache line and then another, prefetch the next cache line before doing the work on the first. This overlaps the work and the prefetch. Prefetch hides memory latency.

The prefetch scheduling distance for memory is about 20-25 instructions. The prefetch scheduling distance for L2 is about 6-10 instructions. These are very rough guides.

Worst Case Scenarios

Avoid worst case scenarios: for associative cache memory, if addresses differ by a power of the cache line size, 64, 128, 256, ..., they are constrained to use fewer cache lines. Each increase in the exponent halves the number of cache lines available, down to the minimum of the cache set size. Similarly, on an associative TLB, if addresses differ by a power of the page size, 8192, 16384, ..., they are constrained to use fewer TLB slots with each increase in exponent.

False Sharing - avoid updating a cache line shared between two or more CPUs.

Constraints - bias the allocation of small objects by a multiple cache line offset to avoid hot cache lines. The slab allocator does this. For an array of structures, use odd multiples of the cache line size.

Coloring - if necessary, allocate physical memory pages in color order. They won't compete for cache.

If you need uniquely colored pages in the kernel or in a module, use `__get_dma_pages(gfp_mask, order)` to allocate up to 32 pages, usually 128K. They will be physically contiguous and therefore in color order.

General

Cache Policy	Cache Miss	Description	Suggestion
load	compulsory	first access	align, pack and prefetch
placement	conflict	access mismatched to cache design	avoid worst cases
replacement	capacity	working set is larger than cache size	avoid cache pollution
store	combine	single store or large streaming store	all of the above, combine writes, use non temporal instructions

Acknowledgements

Scott Maxwell, David Sifry and Dan Sears suffered through and improved earlier drafts.

References

[Bonwick94] Jeff Bonwick, The Slab Allocator: An Object-Caching Kernel Memory Allocator, Usenix Summer 1994 Technical Conference, 1994.

[Bryant00] Ray Bryant and Bill Hartner, Java technology, threads, and scheduling in Linux: Patching the kernel scheduler for better Java performance, IBM Developer Works, January 2000.

[Dogan99] Cort Dogan, Paul Mackerras and Victor Yodaiken, Optimizing the Idle Task and Other MMU Tricks, Third Symposium on Operating Systems Design and Implementation, 1999.

[Handy98] Jim Handy, The Cache Memory Book, 2nd edition, Academic Press, 1998.

[HennPatt96] John Hennessy and David Patterson, Computer Architecture, 2nd edition, Morgan Kauffman, 1996.

[Honeyman99] Peter Honeyman et al, The Linux Scalability Project, University of Michigan CITI-99-4, 1999.

[Ingo00] Ingo Molnar, <http://people.redhat.com/mingo/mmx-patches/>

[Intel99a] Intel Architecture Optimization Reference Manual, developer.intel.com, 1999.

[Intel99b] Block Copy Using Intel Pentium III Processor Streaming SIMD Extensions, developer.intel.com, 1999.

[Lynch93] William L. Lynch, The Interaction of Virtual Memory and Cache Memory, Stanford CSL-TR-93-587, 1993.

[Maxwell99] Scott Maxwell, Linux Core Kernel Commentary, Coriolis, 1999.

[McKusick96] Kirk McKusick et al, The Design and Implementation of the 4.4BSD Operating System, Addison Wesley, 1996.

[Schimmel94] Curt Schimmel, UNIX Systems for Modern Architectures: Symmetric Multiprocesssing and Caching for Kernel Programmers, Addison Wesley, 1994.



[Seward] Julian Seward, Cacheprof, www.cacheprof.org

[Shanley97] Tom Shanley, Pentium Pro and Pentium II System Architecture: 2nd edition, Mindshare, 1997.

[Stallman00] Richard Stallman, Using and Porting GNU CC: for version 2.95, Free Software Foundation, 2000.

[Strunk99] Strunk and White, The Elements of Style, 4th edition, Allyn and Bacon, 1999.

*This paper was originally published in the
Proceedings of the 4th Annual Linux
Showcase and Conference, Atlanta, October
10-14, 2000, Atlanta, Georgia, USA*

 [Papers Index](#)
 [USENIX home](#)

Last changed: 8 Sept. 2000 bleu